Counting Dependence Predictors

Undergraduate Honors Thesis

Franziska Roesner
Department of Computer Sciences
The University of Texas at Austin
franzi@cs.utexas.edu

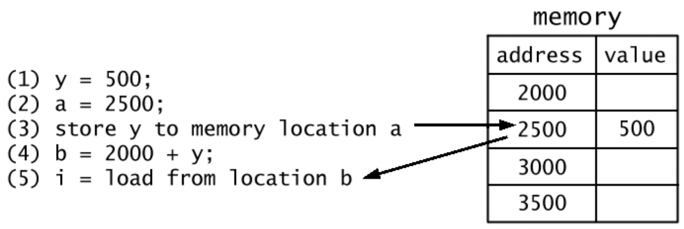
Supervising Professor: Doug Burger
Second Reader: Stephen W. Keckler

May 2, 2008

Abstract

Modern processors rely on memory dependence prediction to execute load instructions as early as possible, speculating that they are not dependent on an earlier, unissued store. To date, the most sophisticated dependence predictors, such as Store Sets, have been tightly coupled to the fetch and execution streams, requiring global knowledge of the in-flight stream of stores to synchronize loads with specific stores. This thesis proposes a new dependence predictor design, called a Counting Dependence Predictor (CDP). The key feature of CDPs is that the prediction mechanism predicts some set of events for which a particular dynamic load should wait, which may include some number of matching stores. By waiting for local events only, this dependence predictor can work effectively in a distributed microarchitecture where centralized fetch and execution streams are infeasible or undesirable. I describe and evaluate a distributed Counting Dependence Predictor and protocol that achieves 92% of the performance of perfect memory disambiguation. It outperforms a load-wait table, similar to the Alpha 21264, by 11%. Idealized, centralized implementations of Store Sets and the Exclusive Collision Predictor, both of which would be difficult to implement in a distributed microarchitecture, achieve 97% and 94% of oracular performance, respectively.

Contents



```
(1) y = 500;
(2) a = 2500;
(3) store y to memory location a
(4) b = 2000 + y;
(5) i = load from location b
```

Figure 1. *This example code illustrates the memory disambiguation problem. Instructions 3 and 5 access the same memory location, but this cannot be known a priori, since the memory locations must first be computed by other instructions.*

1 Introduction

Microprocessor performance has been increasing exponentially for the past 30 years, one of the key enabling technologies that has powered the computing revolution. Unfortunately, continued improvement at historical rates is far from guaranteed, since many technical challenges stand in the way of continued progress. One of these challenges is *memory disambiguation*. In modern processors, there are many operations in flight at once, including reads and writes to memory, which can be executed out of order. It is unknown in advance which reads and writes will access the same memory location. If a load reads from memory before an earlier store writes to the same location, the load may have read an incorrect value and the processor's pipeline must be flushed, resulting in performance degradation. Figure 1 illustrates the memory disambiguation problem.

Memory disambiguation was an enormous problem for dataflow architectures in the 1970s/80s, which required that all instruction dependences be known statically, and likely prevented their adoption by forcing them to require unconventional languages. Until now it has been one of the primary impediments to scaling the performance of high-performance, single-threaded microprocessors. Therefore, modern processors rely on *memory dependence prediction* to execute load instructions as early as possible: they predict which reads to memory should wait for a write that will match, and which are safe to issue early. Because most loads are dependent on one or no stores, it is not necessary for them to wait on the completion of all previous stores. Dependence predictors rely on the previous execution history of a load to make predictions intended to minimize performance losses due to loads executing too early (and therefore causing flushes) or to loads being held back longer than necessary (and therefore losing valuable opportunities for parallelism).

The early work on dependence predictors began with Moshovos and Sohi's PC-matching predictor [10] and with the 21264 load-wait table [3]. Chrysos and Emer's Store Sets predictor [2] achieves close to ideal performance, defined as each load waiting only for the exact stores, if any, that will forward values to the load.

Some key assumptions under which previous dependence predictors were shown to be near-ideal have changed. Global wire delays have resulted in the emergence of partitioned architectures, such as CMPs and tiled architectures [24]. Distributed architectures that execute single-threaded code [5, 14, 20, 23, 24] without a single centralized fetch and/or execution stream will make it challenging to deploy predictors such as Store Sets, which require observation of a complete and centralized stream of fetched instructions to synchronize loads with specific stores. Previously proposed dependence prediction mechanisms also rely on global execution information to track the completion of stores that trigger the wakeup of deferred loads. Furthermore, these distributed architectures, with heavily partitioned and distributed data cache banks, may benefit from placement of the dependence predictors at the cache banks (*memory side* predictors), even at the cost of a slight reduction in accuracy over *execution side* predictors. These factors result in a need for new dependence predictors that work effectively for large-window distributed microarchitectures.

This thesis proposes a class of dependence predictor designs, called *Counting Dependence Predictors* (CDPs). CDPs are designed to work well in distributed architectures, in which a centralized fetch stream and access to some global execution information may be infeasible. Dependence predictor designs must therefore strive to make accurate predictions with as little remote information as possible. Any needed information must be available, or easily made available, locally to the predictor. The enabling feature in CDPs is that the prediction mechanism is oblivious of the fetch stream and predicts the local events for which a particular dynamic load should wait. These events include some number of matching stores and can be tracked without complete global execution information. In one implementation of CDPs, a PC-indexed table of counters will produce a state that indicates the number of matching stores for which a dynamic load should wait: zero (aggressive), N (already arrived or arriving later), or all of them (conservative). Because any earlier store to the same address is considered a match for a load, rather than waiting on some specific store that was fetched, the predictor mechanism is decoupled from reliance on the fetch stream. If deferred loads are held at their cache bank, information about matching stores will be available locally.

This thesis evaluates CDPs in the context of the TFlex microarchitecture [8], a fully distributed tiled architecture that supports an issue width of up to 64 and an execution window of up to 4,096 instructions. Since control decisions, instruction issue, and dependence prediction may all happen on different tiles, a distributed protocol for handling efficient dependence prediction is necessary. This thesis describes such a protocol and shows how distributed dependence prediction can be efficiently run on an aggressive processor with only small losses in performance (1-2%) over an ideal, centralized CDP with no routing latencies.

The ideas behind CDPs are applicable to any architecture with distributed fetch and distributed memory banks, in which the comprehensive event completion knowledge needed by previous dependence predictors is costly to make available globally. This thesis describes a specific control protocol implementation for TFlex, but this implementation may differ for other architectures with different state.

The best-performing CDP configuration achieves 92% of oracular performance, showing only small performance drops due to the routing latencies of the distributed prediction protocol. As a point of comparison, a distributed load-wait table, similar to the Alpha 21264, achieves 81% of ideal performance. Idealized, centralized implementations of Store Sets and Yoaz et al.'s Exclusive Collision Predictor achieve 97% and 94% of oracular performance, respectively. Of these predictors, only Load-Wait is straightforward to implement in a distributed environment, as it is essentially a degenerate form of the CDP. Although remaining performance can be mined from large-window distributed dependence prediction, the CDP designs evaluated in this thesis outperform Load-Wait by 11%.

2 Related Work in Dependence Prediction

According to Onder's proposed classification [13], dependence predictors are typically independence predictors that predict zero matches very well, pair predictors that are tuned for predicting exactly one matching store, and set predictors that aim to capture more intricate load/store patterns. Counting Dependence Predictors are a hybrid that can switch between these classes of predictors depending on the program workload.

All previous work in dependence prediction has relied on a central point of fetch to build tables, and/or the ability to observe a centralized execution stream to track events needed to wake deferred loads. Predicting loads to be dependent on specific stores requires knowing which stores are in flight and when they complete, and thus observing centralized fetch and execution streams, which becomes infeasible or undesirable for large-window distributed architectures. These requirements make it difficult to distribute the predictors efficiently.

Early work on dependence predictors by Moshovos and Sohi identified the potential of memory speculation for out-of-order processors. They proposed a predictor that identified recurring RAW memory violations using two content-addressable memory (CAM) tables [10], one for static and the other for dynamic load-store pairs. Entries, consisting of the PCs of a static load-store pair, are allocated in the memory dependence prediction table (MDPT) when a load violation occurs. When a memory instruction executes, if it finds a MDPT entry, it finds or allocates an entry in the sychronization table (MDST), which consists of dynamic load-store pairs identified by a unique instance number. A load with an active entry (or entries, if multiple dependences are supported) must wait for the corresponding store(s) to complete before executing. This scheme requires the ability to track the completion of every store globally, which is difficult to implement efficiently in a distributed environment, where stores may map to different processing tiles.

Moshovos and Sohi's later work [11] uses a prediction scheme that assigns a common tag to all dependences that have common producers (stores) or consumers (loads). The tag is used to identify all of these dependences collectively, and the correct association between a load and a store is enforced based on which store is in flight. This mechanism is similar to Chrysos and Emer's Store Sets predictor, which identifies sets of matching loads and stores, and makes dependent loads wait on particular dependent stores [2]. The Store Sets implementation consists of two tables, the store set ID table (SSIT) and the last fetched store table (LFST). When a load is fetched, it acquires a store set ID from the SSIT and uses it to access the LFST, which outputs the most recently fetched store in that set, upon which the load is made dependent. When stores are fetched, they also access the SSIT and LFST, serializing stores in the same set. To handle multiple dependences between different load and store PCs, store sets are merged if a violation occurs involving a load or store that has already been assigned an SSID. Both of these schemes require observation of the fetch stream to build up the prediction tables. Loads depend on specific older stores that are in flight, and these dependences are marked as the stores are fetched. Access to global execution information is also necessary to track the completion of stores.

Yoaz et al. developed a much simpler but still effective predictor based on distances between dependent loads and stores [25]. In its simplest form, their collision history table (CHT) works like a load-wait table, holding back loads predicted dependent until all older stores have completed. The inclusion of dynamic distances between a load and the store with which it collides allows loads to be advanced past some but not all stores in flight. The distance with which the load's CHT entry is annotated will converge to the smallest distance seen as the load violates with other stores. The distances are based on load and store ages, which are generally stamped at fetch, making it difficult to support distributed fetch. This predictor also poses a challenge for distributed execution, as the completion of each store must be tracked to determine when all stores a given distance away from a deferred load have completed.

Several researchers have adapted these designs. Notably, Sha, Martin and Roth enhanced the Store Sets predictor with path based information and proposed training on both violations and forwardings [16]. Similarly, Subramaniam and Loh extended the distance predictor with partial tags and confidence estimates to improve its accuracy even further [22]. Other follow-on work has included several LSQ optimizations [16, 17, 21, 22] and direct load-store communication [12].

Bench-	No Matches		One Match		Two+ Matches	
mark	static	dynamic	static	dynamic	static	dynamic
bzip2	64.2	93.3	20.8	6.7	15.1	0.0
crafty	81.4	95.8	14.5	4.1	4.1	0.1
gcc	79.4	99.9	15.2	0.1	5.3	0.0
gzip	72.3	92.1	20.0	7.2	7.7	0.7
mcf	71.1	98.2	22.3	1.8	6.6	0.0
parser	79.4	90.7	14.0	8.5	6.6	0.8
perlbmk	79.8	86.8	18.4	12.8	1.8	0.4
twolf	88.8	95.8	8.9	4.2	2.3	0.0
vortex	80.6	90.3	16.1	9.5	3.2	0.2
applu	78.1	87.5	21.2	12.5	0.7	0.0
apsi	90.4	96.7	9.0	3.3	0.6	0.0
art	96.8	99.8	2.7	0.2	0.5	0.0
mesa	82.2	93.5	15.7	5.7	2.2	0.9
mgrid	85.5	98.9	13.1	0.4	1.4	0.6
sixtrack	77.1	90.2	20.5	9.4	2.4	0.4
swim	100.0	100.0	0.0	0.0	0.0	0.0
wupwise	77.5	25.3	19.1	62.1	3.2	12.6
average	81.5	90.3	14.8	8.7	3.8	1.0

Table 1. *Breakdown (Percent) of Store Matches for Static and Dynamic Loads in SPEC2000 benchmarks: Most loads conflict with one or fewer in-flight stores. This and table 2 were generated with a 16-core TFlex configuration, with up to 2048 instructions in flight, 512 of which can be memory instructions.*

2.1 Applicability to Distributed Architectures

Though this thesis describes a distributed CDP protocol tailored specifically for the TFlex microarchitecture, other designs can also benefit from the advantage of the CDP [4, 5, 19, 20, 23, 24]. For example, the protocol described in Section 5 can easily be adapted for Core Fusion [5] by giving its steering management unit (SMU) the responsibilities of the controller core. While Ipek et al. describe how a Store Sets implementation would be possible [5], their preference for the simplicity of per-core load-wait tables is a testament to the difficulty of distributing a predictor that requires information not easily or cleanly made globally available.

In addition, while the block-atomic nature of the ISA used by TFlex simplifies some components of the protocol, this technique could be employed with other ISAs by artificially creating blocks from logical blocks in the program for the sake of simplified store completion tracking. Blocks provide simplicity advantages because they allow operations like tracking completed stores and determining if a given instruction is in flight to be done on a block granularity and because they naturally separate instructions into groups that may have distributed control points.

3 Counting Dependence Prediction

Counting Dependence Predictors predict the events for which a particular dynamic load should wait. These events may include some number of arbitrary matching stores, rather than specific stores identified before execution. This section presents data that indicates that it is possible to predict how many in-flight stores a load will conflict with and a possible CDP implementation that predicts loads to wait for zero, one, or more store matches.

Bench- mark	no match	one match	two+ match	0,1 flip	1,2+ flip	0,2+ flip	0,1,2+ flip
bzip2	67.6	0.0	0.0	8.8	0.0	0.0	23.5
crafty	82.2	0.3	0.0	12.5	0.2	0.0	4.7
gcc	81.1	1.2	0.0	11.1	0.3	0.0	6.3
gzip	72.4	0.0	0.0	17.1	0.2	0.0	10.3
mcf	68.6	0.0	0.0	22.1	0.0	0.0	9.2
parser	82.3	0.0	0.0	9.4	0.0	0.0	8.3
perlbmk	77.3	1.4	0.0	19.2	0.0	0.0	2.2
twolf	90.0	0.1	0.0	7.3	0.0	0.0	2.6
vortex	80.1	1.5	0.0	14.4	0.5	0.2	3.2
applu	73.1	0.9	0.0	25.0	0.0	0.0	0.9
apsi	90.0	0.0	0.0	9.2	0.0	0.0	0.7
art	97.3	0.8	0.0	1.4	0.3	0.0	0.3
mesa	81.0	0.2	0.0	16.2	0.0	0.0	2.6
mgrid	84.9	1.4	0.0	12.0	0.0	0.0	1.6
sixtrack	73.6	0.6	0.0	22.7	0.0	0.0	3.1
swim	100.0	0.0	0.0	0.0	0.0	0.0	0.0
wupwise	75.6	1.0	0.0	19.0	0.0	0.0	4.4
average	81.0	0.6	0.0	13.4	0.1	0.0	4.9

Table 2. *Breakdown (Percent) of Dynamic Behavior of Static Loads in SPEC2000 benchmarks: Most static loads never conflict with any in-flight stores across their dynamic instances; if they do, they usually flip between zero and one store match.*

3.1 Store-Load Dependence Behavior

Table 1 shows a breakdown of the number of in-flight matching older stores for each load, measured with an execution window of up to 512 memory instructions. Static loads are identified uniquely by their PC. A given static load may be executed more than once, and the *dynamic* columns refer to these dynamic instances of static loads. For example, 72.3% of gzip's static loads have no store matches in at least some of their dynamic instances, but 92.1% of the dynamic instances during the execution of the program conflict with no stores. Most load instructions conflict with no in-flight stores and can safely be executed as soon as their address is available. Of the loads that must wait for data from one or more stores before executing, most depend on only one in-flight store. A minority of static loads (3.8% on average), and even fewer dynamic instances (1.0% on average), must wait for two or more stores before executing safely.

Table 1 does not distinguish among loads that have different behavior across dynamic instances. Table 2 shows a breakdown of the dynamic behavior of static loads. Each percentage indicates what fraction of static loads have dynamic instances that exhibit the behavior described by that column. For example, 72.4% of gzip's static loads match with no stores every time they are executed, and 17.1% of the static loads dynamically alternate between zero and one matching stores. According to these data, most static loads will never alias with any in-flight stores and thus each dynamic instance of that load can safely be executed as soon as its address is available. Loads that flip between different numbers of store matches are less predictible for any dependence predictor but may nevertheless be grouped into useful categories.

These data indicate that it is beneficial to predict when it is safe to execute a given load by predicting for how many store matches that load should wait. Counting Dependence Predictors wait for a learned number of stores to complete before waking a load predicted to be dependent. Unlike many previous dependence predictors, CDPs do not predict dynamic loads to be dependent on one

```
#define SIZE 100

void main()
{
   int x = 0; int y, k, i;

   int A[2*SIZE];
   int B[SIZE];

   for( i = 0; i < SIZE; i++ )
   {
      y = i;
      if( i % 2 == 0 )
          k = i;
      else
          k = i + SIZE;

      A[y] = y; //Store A
      A[k] = k; //Store B
      x += A[i]; //Load C
   }
}
```

Case #1: Two Matches

i = 2

Execution order:

Load C (from i)
Store A (to i)
Store B (to i)

Case #2: One Match

i = 3

Execution order:

Load C (from i)
Store A (to i)
Store B (to i + SIZE)

Case #3: Early Match

i = 3

Execution order:

Store A (to i)
Store B (to i + SIZE)
Load C (from i)

Figure 2. *Load C matches different numbers of stores in different cases. In the first two cases, unless the load waits long enough, a violation will occur because the load executes too early. In the last case, the matching store executes correctly before the load.*

or more specific dynamic stores, but rather on a predicted number of arbitrary stores.

A load violation occurs when a load executes before an older store (earlier in program order) to the same address. When such a violation is detected, the pipeline must be flushed. Figure 2 shows how a given static load may conflict with a different number of stores dynamically. In the code given, Load C follows Stores A and B in program order. Load C will always be dependent on Store A, but whether or not it is also dependent on Store B depends on the value of *i*. The three cases in Figure 2 show different ordering possibilities during the execution of the code. The states of one possible CDP, outlined in Table 3, are designed to handle all of these cases and to transition among them.

3.2 Prediction Types

When a load is predicted dependent, it must be awakened by some triggering event, as defined by the predictor. Various information, such as the control path, the load's PC, or its address can be used to predict which event should cause a load to issue. In a distributed architecture, this information would ideally be either locally available or globally broadcast for other purposes. CDPs aim to use as little additional remote messaging as possible to predict the type of event that should cause a load to be woken.

The states of one possible CDP are outlined in Table 3. Different prediction types are defined by the event type that triggers the load wakeup:

1. An *aggressive* load can execute speculatively as soon as its address is available.

State	Event waiting for
Aggressive	None
Conservative	Completion of all previous stores
N-store	N matching stores arriving before or after load

Table 3. *Overview of CDP States*

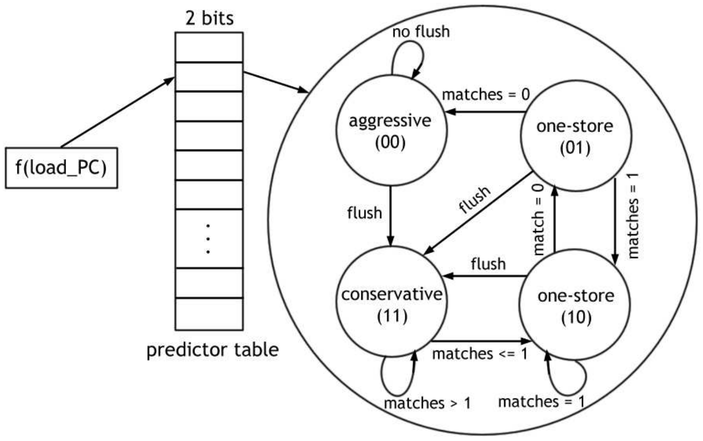


Figure 3. *CDP Dependence Table and State Machine: A load hashes into the predictor table with its PC, interpreting the value found there as one of the states shown. The states are updated based on load behavior.*

2. *Conservative* loads must wait until all previous stores (in program order) have completed.
3. *N-store* loads wait for a learned number of arbitrary matching older stores. In the implementation described here, loads predicted in this third category wait on any one store match (i.e. N equals one). Because the load's address must be resolved before store matches can be counted, the load is issued, routed to the corresponding cache bank, and buffered until its wakeup event happens.

The third type of prediction raises the question of what constitutes a store-match event and whether using the timing of matching stores can help to narrow false matches. There are two possibilities for defining a store-match event; the assumption in the protocol description above is that a store match happens when a store to the same address resolves after a waiting load. However, the particular store on which a load is dependent may resolve before the load instead. Therefore, I also evaluated a second policy, called *already arrived stores*, in which loads that are predicted to be dependent on one store are woken immediately if a matching store still in flight has already resolved. Waking one-store loads based on the presence of an already issued older store that is likely to be the load's only store match reduces the number of costly cases in which a load is incorrectly predicted one-store and needlessly waits for all older stores to complete. Considering early arriving stores wakes one-store loads and trains the dependence predictor based on store-to-load forwardings.

3.3 Wakeup and Training Policies

The predictor is a simple table hashed by load PC that contains 2-bit values representing one of the three states described in the previous subsection. The table is initialized with each entry in the aggressive state and is updated according to subsequent load behavior, as shown in Figure 3. The following describes the behavior of each state:

- *Aggressive*: If a load was issued aggressively but should have waited for an older store, a dependence violation flush is triggered. The load's corresponding predictor table entry is set to conservative. Otherwise the prediction was correct and this entry in the table remains in the aggressive state.

- *Conservative*: As a load predicted conservative waits for all older stores to complete, the number of older stores that execute and conflict with the deferred load are counted, and the corresponding entry in the prediction table is updated to one-store if the count shows the conservative execution to have been overkill (i.e. fewer than $N + 1$ store matches were counted). Otherwise this entry in the table remains in the conservative state.

- *N-store*: In a basic CDP implementation, when a load is predicted N-store, the number of older matching stores that execute while the load waits are also counted, and when this number reaches N, the load is woken up to issue to memory. If the number of actual store matches does not reach N, the load is effectively treated as conservative since it must wait needlessly for all older stores to complete. In this case, the dependence predictor is de-trained. If instead the load was not held back long enough and a violation occurs due to a second store match, a flush is triggered and the corresponding table entry is set to conservative. The presence of two one-store states in the implementation described here adds hysteresis and decreases the sensitivity of the predictor. I use this version of N-store as the baseline for the evaluation in this thesis, but these two states could represent separate predictions of some number of stores or some other event used to trigger load wakeup. Section 3.5 discusses slightly different treatments of the N-store state.

3.4 CDP Advantages and Disadvantages

The CDP performs well when the number of stores that match with a load is consistently small (one, in the case of the CDP implementation described here, but as mentioned, other values of N could be used and/or additional intermediate states could be added). When the CDP correctly predicts that a load will alias with one store, the load is made to wait for only that one store. The CDP performs well especially if the one store with which a load matches is not always the same static store, since it does not predict dependence on a specific store. Other predictors will often force the load to wait longer than necessary: Load-Wait forces the load to wait for all older stores to complete, ECP forces the load to wait for most older stores to complete, and Store Sets grows more conservative as the in-flight store set grows larger. Since most dependent loads are dependent on only one in-flight store, the CDP has this advantage in a significant number of cases.

However, in some cases the CDP is at a disadvantage compared to other predictors. There are two types of dependence mispredictions, too conservative and too aggressive, and certain patterns of load behavior aggravate the CDP's mispredictions in these categories. In particular, loads that dynamically alias with different numbers of stores can cause the prediction states to fluctuate, alternately causing too aggressive and too conservative mispredictions. The following section describes modifications to the basic protocol that address these cases.

Even correct conservative predictions are more costly for the CDP than for more precise predictors like Store Sets. Beyond the N-store state, the CDP does not differentiate between $N + 1$ and all stores. Thus, if a load is dependent on some intermediate number of in-flight stores, Store Sets will hold the load back only until all stores in the in-flight store set have completed, while the CDP will hold the load back until all in-flight stores have completed. Because the load generally does not match with most of these stores, the load may be held back too long.

As an example, Figure 2 shows some of these cases. In the second case, the CDP will perform very well when it predicts correctly, since it will wake the load immediately after the single store match. In the third case, the CDP variant which uses already arrived stores will perform well, since it will let the load execute immediately, rather than waiting for another match that will never happen. In the first case, however, the CDP may not perform as well, even if it correctly predicts the load to be conservative, since it will wait for all older stores (not shown in the example) to complete, rather than deferring the load only until the completion of the two stores with which it actually matches.

3.5 CDP State Machine Optimizations

When the number of matching stores varies among dynamic instances of a given static load, the CDP is at a disadvantage, because the predictor state may fluctuate based on the repeated mispredictions and subsequent updates of the table. I experimented with several modifications to determine what information may help the predictor identify the correct number of stores in such cases.

Specifically, a load might alternate between being dependent on zero or one store(s), causing an unnecessarily conservative load execution half of the time and a violation the other half. Similarly, a load might alternate between being dependent on one or two (or more) stores.

To address the 0-1 case, I modified the CDP to record some bits of the store's PC when a load violates. When the next instance of this load is predicted one-store, the predictor checks if an older instance of the offending store is in flight. If not, the load is allowed to issue aggressively, assuming it will not alias with another static store. This policy aims to reduce the cases in which an independent load is predicted one-store and defaults to waiting for all older stores to complete because no store match ever occurs. This optimization requires additional space (for the bits of the store PC) and can also cause incorrect predictions in the less common case where the load's next dynamic instance is dependent on a different static store.

I address the 1-2 case in a similar way. When a matching store prompts the wakeup of a load predicted one-store, a check is done to see if there are any stores with the same PC in flight between the store match and the load. If so, the wakeup of the load is deferred. This policy approximates the aspect of Store Sets which serializes all in-flight stores belonging to a given store set and makes the load dependent on the last of these stores. This optimization does not require additional storage area, but may in some cases needlessly delay the load's execution.

4 TFlex

I simulate and evaluate CDPs on the TFlex microarchitecture [8], a Composable Lightweight Processor (CLP), that allows simple cores to be aggregated together dynamically. TFlex is a fully distributed tiled architecture of 32 cores, with multiple distributed load-store banks, that supports an issue width of up to 64 and an execution window of up to 4096 instructions with up to 512 loads and stores. Since control decisions, instruction issue, and dependence prediction may all happen on different tiles, a distributed protocol for handling efficient dependence prediction is necessary. Here I give necessary background information about the TFlex architecture upon which the protocol of the next section is based.

The TFlex architecture uses the TRIPS Explicit Data Graph Execution (EDGE) ISA [1] which encodes programs as a sequence of blocks that have atomic execution semantics, meaning that control protocols for instruction fetch, completion, and commit operate on blocks of up to 128 instructions.

The TFlex microarchitecture has no centralized microarchitectural structures. Structures across participating cores are partitioned based on address. Each block is assigned an owner core based on its starting address (PC), instructions within a block are partitioned across participating cores based on instruction IDs, and the load-store queue (LSQ) and data caches are partitioned based on load/store data addresses.

A block is distributed across the I-caches of all participating cores. The block owner core is responsible for initiating fetch and predicting the next block. Once predicted, the next-block address is sent to the owner core of the predicted next block. When a memory instruction executes, it is sent to the appropriate core's cache bank based on its target address. Pipeline flushes due to misspeculations are also initiated by the owner of the block causing the misspeculation. Since loads and stores to the same address will always go to the same memory core, dependence violations are detected by the load-store queue at that cache bank. Before committing the block, the owner core must receive completion confirmations of stores, register writes, and one branch from all participating cores. Once the block is ready to commit, the owner sends a commit message to each participating core and waits for acknowledgements. All control, data request and response, and operand communication among cores uses a number of two-dimensional wormhole-routed meshes.

Each block owner has the PCs of all in-flight blocks available. This information allows the 0-1 and 1-2 flip CDP optimization described in the previous section to be implemented efficiently by simply checking whether another in-flight block has the same block PC as the block of the store in question, rather than needing to perform the more difficult and non-centralized operation of determining which stores are in flight. This also prevents mispredictions due to overly specifying stores, since one block usually contains several stores.

5 A Distributed CDP Protocol

A correct protocol for dependence prediction must fulfill several requirements. First, all loads and stores to the same address must be matched. For each load, a prediction must be made and stored, and if the load is deferred, the corresponding wakeup event must be detected and the waiting load must be notified. Thus, the protocol must detect when all stores older than a given load have completed. Finally, the correctness of speculation must be confirmed.

Several of these requirements are non-trivial to implement in a distributed environment. Because instructions can execute on any core, it may to difficult to detect wakeup events, such as the completion of all stores older than a given load. Though all loads and stores to the same address will eventually arrive at the same cache bank in TFlex, it is less trivial to track the completion of older stores to different cache banks. Confirming correctness of speculations is also non-trivial, as significant events may happen elsewhere in the architecture.

Because CDPs use as little information as possible to make predictions–in particular, they do not need to follow all stores in the fetch stream–they are more amenable to operation in a distributed environment. A number of additional control messages, described in this section, are required for correct handling of the issues discussed above.

There are three goals to consider when designing a distributed protocol: few control messages, few control message types (i.e., low protocol complexity), and low latency on the critical path. The distributed CDP protocol I describe achieves all of these goals.

5.1 Distributed Protocol

Figure 4 lists the message types and stages of prediction distributed among different processing cores. The protocol requires four message types, including three not in the base TFlex design. The prediction and wakeup of a load are handled by the protocol as follows. Each of these operations may occur on any core, including on the same core.

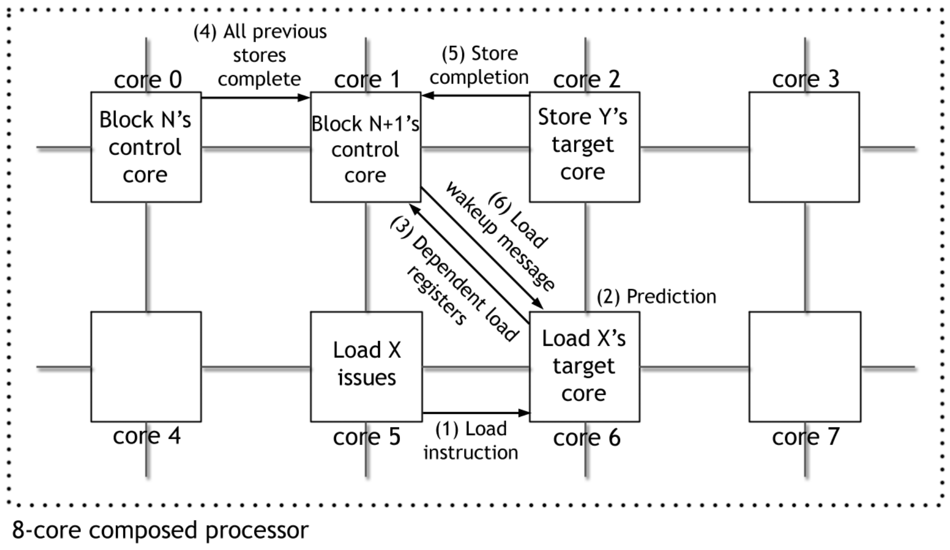


Figure 4. *Distributed Counting Dependence Predictor Protocol: Simple control messages between processing cores are used to implement dependence prediction.*

1. A load is issued at one core (core 5 in Figure 4), and is routed to the core containing the appropriate cache bank, determined by the address of the load.

2. Prediction occurs at the core containing that cache bank (core 6, in this example). If a load is predicted aggressive, it executes immediately. If it is predicted to be dependent (either conservative or waiting on some events), a *registration message* is sent to the controller core, the block owner of the load's block (core 1). The registration message is a request to the block owner to inform the load when all older stores have completed.

3. To enable the block's controller core to know when all stores prior to a particular load have completed, two additional types of messages are needed. First, whenever a store in the block completed, a *store completion message* is sent from the core containing its cache bank back to the controller core. Store completion messages do not need to be added specifically for the purpose of dependence prediction, as they are already necessary for determining block completion.

4. Before a registered load can be safely woken, the controller core must know that all stores older than that load have completed. It is not sufficient to know that all older stores in the load's block have completed, since there may be pending stores in older blocks. Thus, an *all-stores-completed message* is needed, which block owner N sends to block owner $N + 1$ as soon as all of the stores in block N have completed. This single message sent between controller cores of successive blocks prevents the need to broadcast store completion messages to every core.

5. The controller core is responsible for sending *wakeup messages* to any load that has registered with it (i.e., any load which was not predicted aggressive). As soon as all stores older than a

Benchmark	Registration	Wakeup	Store Completion
bzip2	0.0871	0.0871	2.9272
crafty	0.2196	0.2196	1.3628
gcc	0.0029	0.0029	5.3457
gzip	0.0606	0.0606	1.2160
mcf	0.0319	0.0319	0.7255
parser	0.0349	0.0349	1.2567
perlbmk	0.1304	0.1304	1.0702
twolf	0.1978	0.1978	1.5402
vortex	0.1586	0.1586	2.2079
applu	0.4303	0.4303	1.6618
apsi	0.1328	0.1328	3.0648
art	0.2424	0.2424	1.5538
mesa	0.2218	0.2218	1.9681
mgrid	0.1317	0.1317	0.9838
sixtrack	0.3892	0.3892	3.0365
swim	0.0000	0.0000	2.9896
wupwise	0.6659	0.6659	2.1654
average	0.1846	0.1846	2.0633

Table 4. *Breakdown of average number of messages sent per block for each message type (across SPEC2000 benchmark run on a 16-core TFlex configuration). Registration and wakeup messages have identical counts because one wakeup message is sent in response to every registration message. All-stores-completed messages are excluded from this table, as exactly one is always sent per block.*

registered load have completed, the controller core sends a wakeup message back to the core containing the cache bank at which the load is waiting.

6. When a waiting load receives a wakeup message, it is free to execute. The wakeup message is required for loads predicted conservative and loads incorrectly predicted N-store (i.e. those which effectively execute conservatively because no store match ever occurs). Because a memory instruction's cache bank is determined by its address, matching stores will always arrive at the core where the load is waiting. Thus, if there were N matches for an N-store load, that load will already have been woken when the wakeup message arrives. In this case, the wakeup message can safely be ignored. If two matching stores arrive in program order after a later dependent load has issued, the first will wake the load and the second will trigger a violation flush.

5.2 Messaging Overhead

One all-stores-completed message must be sent per 128-instruction block, and two messages (registration and wakeup) must be sent for each load predicted to be dependent on unarrived older stores. Loads correctly predicted independent require no messages at all. In our experiments, each load requires the sending of only 0.28 control messages, on average. Table 4 shows a breakdown per message type of average number of messages per block across all supported SPEC2000 benchmarks. Of these message types, store completion messages, which were already necessary in the base TFlex design, are the highest in volume. Registration, wakeup, and all-stores-completed messages are specific to the CDP protocol.

Experiments show that the message latencies have only small effects on overall CDP performance, since most can be hidden by execution. The case when message latency can lead to performance loss is when a load on the critical path is predicted conservative and needs to wait for the wakeup message before knowing that all older stores have completed. For the best-performing CDP configuration, removing the message latencies improves performance by only 1% on average.

Load registration messages are sent to the control tile by each load predicted dependent. These registrations must be stored at the control tile until the load's corresponding wakeup event has been detected and the wakeup message dispatched, thus requiring space to buffer these messages and ways of handling overflow of this space. The performance results for this thesis are all based on infinite buffer space for these messages, but I found that the maximum number of registration messages held by a control tile at one time (across the SPEC2000 benchmark suite with a 16-core TFlex configuration) is 14. Thus a small buffer at each control tile would suffice. In the event of overflow, any of the strategies described in [15] to deal with LSQ overflow would be appropriate. For instance, a flush could be triggered if a registration from a load in the oldest in-flight block arrives at a full buffer. An alternative solution is to simply force all loads that attempt to register at a control tile with a full buffer to execute aggressively.

5.3 Execution vs. Memory Side

The distributed protocol described above implements dependence prediction on the memory side, after a load has been issued and sent to the core containing its cache bank. Loads index into the predictor table at that core. Alternatively, prediction could occur on the execution side, before the load issues. The advantage of this placement is that the table is indexed by the load's PC, rather than a combination of the PC and address. However, execution-side prediction will require a more complex protocol with additional messaging for little gain.

To model the effect of placing the predictor table on the execution side, I approximated execution-side prediction by having all loads index into an ideally centralized predictor table. This idealized experiment improves CDP performance by 2% over the best performing memory-side implementation, but does not model the effects of complicating the distributed protocol or splitting the prediction table by cores.

5.4 Distribution of other Dependence Predictors

Moshovos and Sohi's initial dependence predictor [10] was designed for Multiscalar [19], a distributed architecture in which a single program is divided into a collection of tasks distributed to a number of parallel processing units. However, there are several reasons why this design cannot be adapated for other distributed architecture like TFlex. First, Multiscalar's predictor was implemented as a centralized structure [10]. Moshovos and Sohi do describe how their predictor can be distributed by replicating the CAM tables at every processing tile [10]. However, because this approach requires the broadcasting of information to keep the tables synchronized and to wake loads, it is difficult to scale it efficiently to 8 or 16 nodes. While a more efficient mechanism for distributing this predictor can be imagined, the key issue is that speculation in Multiscalar is only intra- and not inter-task, and intra-task dependences are enforced by the local core, leaving the dependence predictor to deal only with inter-task dependences [9, 19]. This predictor model works well for Multiscalar with its specific tradeoffs, but works less well for regular superscalars, leading to Moshovos and Sohi's later design not specifically targeting Multiscalar [9].

Moshovos and Sohi's later predictor [11] is similar in concept to Store Sets and poses a similar challenge to effective distribution. Store Sets [2], as discussed before, is tightly coupled with the fetch stream. Loads and stores are assigned a store set as they are fetched, allowing loads to be

Parameter	Configuration
Instruction Supply	Partitioned 8KB I-cache (1-cycle hit); Local/Gshare Tournament predictor (8K+256 bits, 3 cycle latency) with speculative updates; Num. entries: Local: 64(L1) + 128(L2), Global: 512, Choice: 512, RAS: 16, CTB: 16, BTB: 128, Btype: 256.
Execution	Out-of-order execution, RAM structured 128-entry issue window, dual-issue (up to two INT and one FP).
Data Supply	Partitioned 8KB D-cache (2-cycle hit, 2-way set-associative, 1-read port and 1-write port); 64-entry LSQ bank; 1031-entry CDP; 4MB decoupled S-NUCA L2 cache [7] (8-way set-associative, LRU-replacement); L2-hit latency varies from 5 cycles to 27 cycles depending on memory address; average (unloaded) main memory latency is 150 cycles.
Simulation	Execution-driven, validated in TRIPS-like configuration to be within 7% of TRIPS prototype hardware cycle counts.
Benchmarks	17 SPEC CPU benchmarks currently supported (9 Int, 8 FP), simulated with single SimPoints of 100 million instructions [18].

Table 5. *Single Core TFlex Microarchitecture Parameters*

made dependent on specific stores and stores within the same set to be serialized. This approach also requires tracking events that may not occur in the same place in the microarchitecture, making distributed execution difficult as well.

Distance predictors such as that of Yoaz et al. [25] require that memory instructions each be assigned relative ages, typically at fetch/decode. Loads are then made to wait on stores a certain distance (in dynamic instructions) away. This scheme requires tracking specific stores in flight. The store in question may not actually be a match and may not report to the core responsible for waking the load. Thus it is insufficient to know that all stores belonging to controller cores of previous blocks have completed, since a load in a given block may be waiting on an arbitrary store in the middle of a previous block. A straightforward implementation requires the broadcasting of store completion information, whereas the CDP requires only one point-to-point message per store.

6 Experimental Results

6.1 Experimental Apparatus

The experiments described in this section were run using a subset of the SPEC2000 benchmark suite (17 SPEC CPU benchmarks currently supported, 9 Integer and 8 FP), simulated with single SimPoints of 100 million instructions [18], on a simulator that models the TFlex microarchitecture. Table 5 details the simulator configuration for one core. Unless otherwise noted, the configuration used for these experiments is 16 composed cores, which corresponds to an execution window of up to 2048 instructions, 512 of which may be memory instructions. The flush penalty modeled requires 5-13 cycles to detect the misprediction, to flush the mispredicted state, and to reinitiate dispatch; some additional cycles are required to refill the pipeline.

6.2 Predictor Configurations

All results are compared to the cycle counts achieved by perfect memory disambiguation, in which loads are made to wait only exactly as long as necessary without causing a violation. As soon as exactly all of the stores (if any) upon which a load is dependent have completed, the load executes. I evaluated the following load execution strategies:

1. *Conservative*: All loads wait until all older stores have completed.

2. *Aggressive*: All loads execute as soon as their addresses are available.

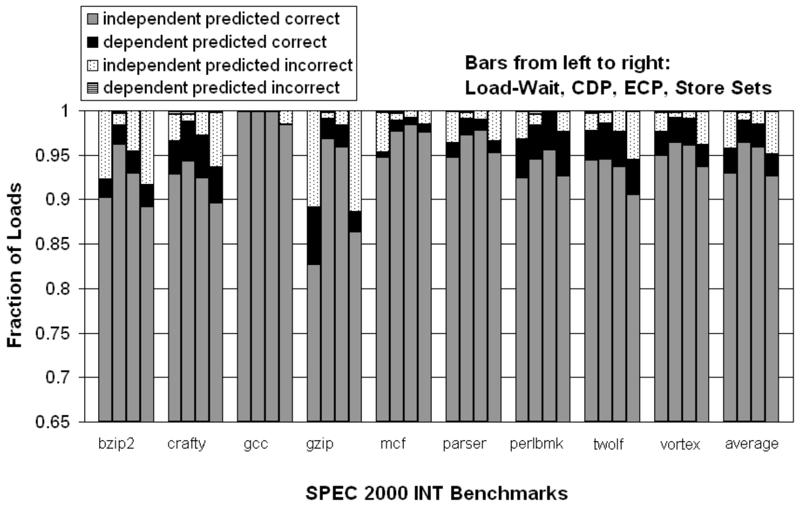


Figure 5. *Breakdown of Predictions for All Loads for SPECINT Benchmarks*

3. *Load-Wait*: A load is predicted either dependent or not; if it is predicted dependent, it waits on the completion of all previous stores. This policy is essentially a CDP with only two options (zero or all stores). The Load-Wait predictor is distributed using the same protocol as described in Section 5. The predictor table is reset every 10000 blocks to prevent overly conservative load execution as the table saturates.

4. *CDP*: I use the best CDP configuration, including all of the modifications described in Section 3. I used a prediction table size of 1031 entries per core, indexed using $LoadPC \bmod (TableSize)$. Using a prime number for $TableSize$ reduces aliasing and allows this modulus to be computed efficiently in hardware [6].

5. *Store Sets*: I implemented a Store Sets predictor according to the description in Chrysos and Emer's paper [2]. This implementation is ideal in that message latencies are not modeled and access to a centralized fetch stream and execution information is assumed. For these experiments, I sized the centralized Store Sets structures to be comparable with the cumulative size of the distributed CDP structures.

6. *Exclusive Collision Predictor (ECP)*: I also implemented a version of Yoaz et al.'s Exclusive Collision Predictor [25]. I use a tagless collision history table (CHT) augumented with distance information. Once a load violates, its entry in the table is marked valid and continues to predict a collision until the table is cleared (every 10000 blocks). The CHT is sized to be comparable to the CDP table, and I use the same hash function to index into it. This implementation is also ideally centralized and message latencies are not modeled.

6.3 Accuracy

The graphs in Figures 5 and 6 show a breakdown of the accuracy of different prediction mechanisms. Each set of bars (per benchmark) shows the breakdown for Load-Wait, CDP, ECP, and Store Sets. Each bar represents the fraction of all loads executed that were correctly predicted

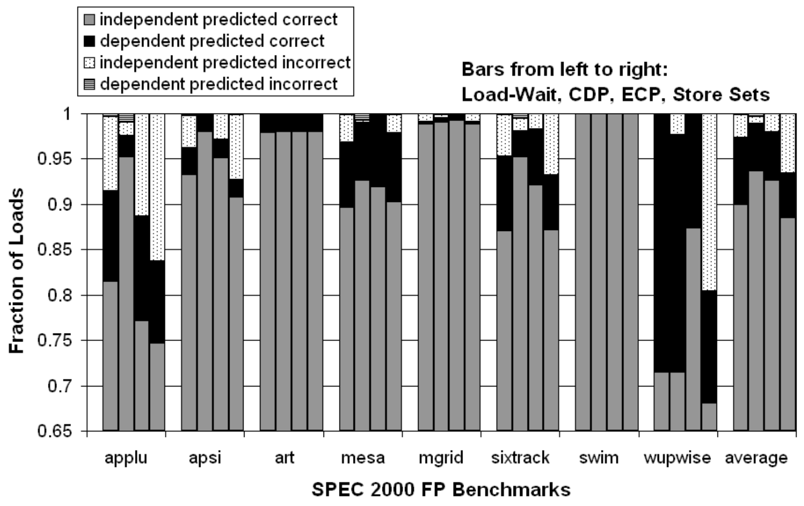


Figure 6. *Breakdown of Predictions for All Loads for SPECFP Benchmarks*

independent, correctly predicted dependent, incorrectly predicted independent, and incorrectly predicted dependent. An incorrectly predicted independent load results in a flush, while an incorrectly predicted dependent load results in later-than-necessary issue of the load.

On average, CDP mispredicts fewer independent loads than do any of the other schemes. By dynamically de-training the predictor rather than requiring an occasional clearing of the table, it avoids becoming too conservative as prediction histories build up.

However, when the CDP does make an overly conservative prediction, it can be more costly than for Store Sets or the ECP. If a load is predicted one-store, but no store match ever occurs, then the load defaults to waiting for all older stores to complete. By contrast, if Store Sets makes an overly conservative prediction, the load will not wait for all older stores, but only the additional stores that have mapped to the same set. ECP will also generally not have the worst case behavior of the CDP in these situations, but is still likely to lose more parallelism than Store Sets. This difference is because Store Sets will not make loads dependent on stores that are not in flight, while the ECP may degenerate to essentially Load-Wait behavior as the predicted dependence distance decreases.

The CDP mispredicts slightly more dependent loads than the other predictors. By having most loads wait only on one matching store, CDP may miss the rarer cases in which there is another match coming that is not caught by the 1-2 optimization. This case will result in a violation flush. By contrast, once a load violates, Load-Wait will force its later instances to execute conservatively until the table is cleared, thus never causing another violation. Store Sets will also always synchronize loads with their previously conflicting stores, preventing reviolations of the same load-store pairs until the table is cleared. The ECP, like the CDP, may fail to prevent reviolations if the control path is different for the next instance of a previously violating load.

Loads that are counted as correctly predicted dependent are not all handled in the same way. For instance, when the Load-Wait strategy forces a dependent load to wait for all older stores to complete, the dependence was correctly identified, but potential parallelism was still lost, since the load only needed to wait for the stores upon which it was dependent, not all stores. In this sense, the CDP may have an advantage over other predictors in the case where a load is correctly predicted dependent. When the CDP correctly predicts a load to wait for one store match, the load is not delayed needlessly past that one match. Since most dependent loads conflict with only one in-flight

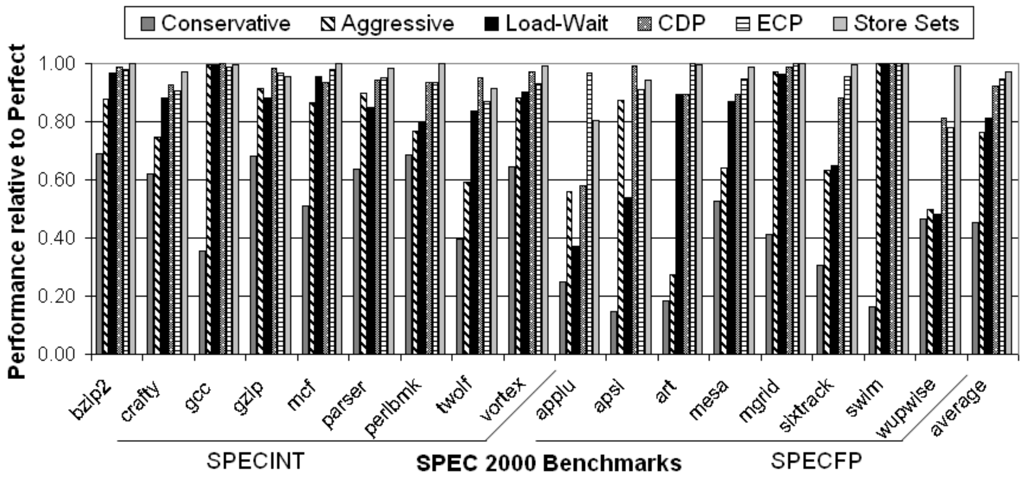


Figure 7. *Comparison of Dependence Prediction Mechanisms*

store, this situation is not rare.

By contrast, ECP will make the load wait for all older stores up to some point specified by the distance, which may degenerate to all older stores in the worst case. Store Sets is less sensitive to this situation since loads wait for specific stores, but as in-flight store sets grow large, loads may be needlessly delayed. In particular, if a load is always dependent on one store, but this store differs across dynamic instances of the load, Store Sets will delay the load needlessly, while the CDP will wake it as soon as the store match occurs.

6.4 Performance

The graph in Figure 7 compares the performance, measured in cycles, of the predictor configurations relative to perfect disambiguation. The best CDP protocol achieves 92% of the performance of perfect, whereas aggressive achieves 76% and Load-Wait achieves 81%. Conservative execution gives by far the worst performance, achieving only 45%. Store Sets achieves 97% of perfect performance and the ECP achieves 94%.

Since most loads match with only zero or one stores (see Table 1), making every load wait on all previous stores (conservative execution) is unnecessary and loses opportunities for parallelism. The Load-Wait strategy only executes a subset of load conservatively, but this approach is still far too conservative, and Load-Wait performs barely better than aggressive execution, despite the high cost of load violation flushes in the aggressive case. The CDP performs 11% better than the Load-Wait policy while only slighly increasing the complexity of the predictor structures.

Ideal Store Sets still outperforms the best CDP configuration by 5% on average. By making loads dependent on a specific dynamic set of in-flight stores, Store Sets can avoid some of the pathological cases that can arise for the CDP in which dynamic instances of a load each have a different number of store matches. The ECP, which outperforms the CDP by 2% on average, will also delay loads for a shorter period of time, in the average case, than does the CDP when it predicts loads conservative.

The CDP does outperform Store Sets and the ECP for several benchmarks, especially those where the CDP mispredicts fewer independent loads. The accuracy breakdown indicates that the CDP's higher misprediction rate of dependent loads, and subsequent violation flushes, degrades the overall

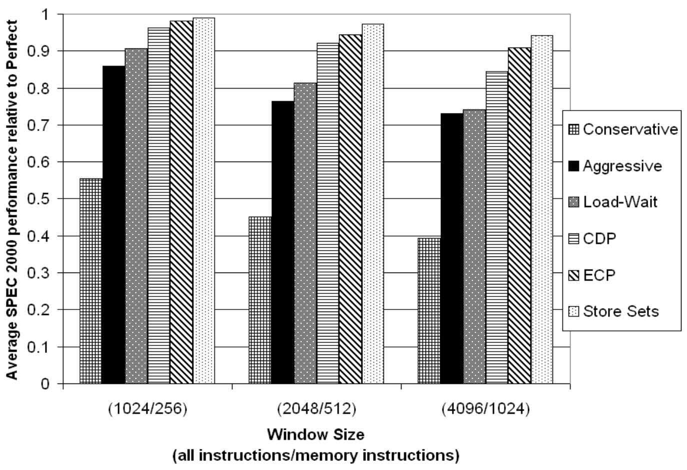


Figure 8. *Predictor Performance with Increasing Window Size*

average performance.

The models of Store Sets and the ECP that I implemented assume access to a centralized fetch and execution stream without additional overhead. A nontrivial amount of overhead (message latencies, broadcasting of global information needed locally by the predictor) would be required to distribute these prediction mechanisms in the same manner as the CDP, since the needed information cannot always efficiently be made locally available. Implementing them as ideally centralized allowed me to implement them as originally described without adding distribution complexity. This approach provides the upper bound on the performance comparing to the CDP, which is designed for a distributed system. Furthermore, I also model an ideal centralized CDP, as mentioned in section 5.3, which achieves on average 94% of perfect, comparable to the ECP.

Figure 8 shows the average performance of the different prediction schemes as the window size increases from an 8-tile TFlex configuration (up to 1,024 instructions, up to 256 of which can be memory instructions, in flight at once) to 16-tile and 32-tile configurations. The larger the window, the more a predictor's performance degrades due to the issues described above. With a window of up to 4,096 instructions, with up to 1,024 memory instructions, the CDP (with message latencies modeled) achieves about 85% of ideal performance. The performance of Load-Wait drops to 74%, since it forces all loads predicted dependent to wait on the completion of all older stores. As the number of memory instructions in flight increases, this policy becomes more costly. At this window size, Store Sets (ideally centralized, no message latencies) still achieves 94% of ideal performance, but the difficulty of supporting a distributed Store Sets-like protocol increases.

6.5 Sensitivity Studies

The graph in Figure 9 shows the performance gain from incrementally adding the modifications described in Section 3. Each bar in the graph includes the modifications added in all of the bars to the left of it, with the exception of 1-2 flip, which does not include the 0-1 flip optimization. *Full* includes both the 0-1 and the 1-2 optimizations. Between the basic CDP protocol and the version including all optimization, there is a 4% performance improvement.

All of the modifications to the CDP protocol (Figure 9) improve or maintain performance across most benchmarks. The 1-2 flip optimization sometimes slightly degrades performance, as it may

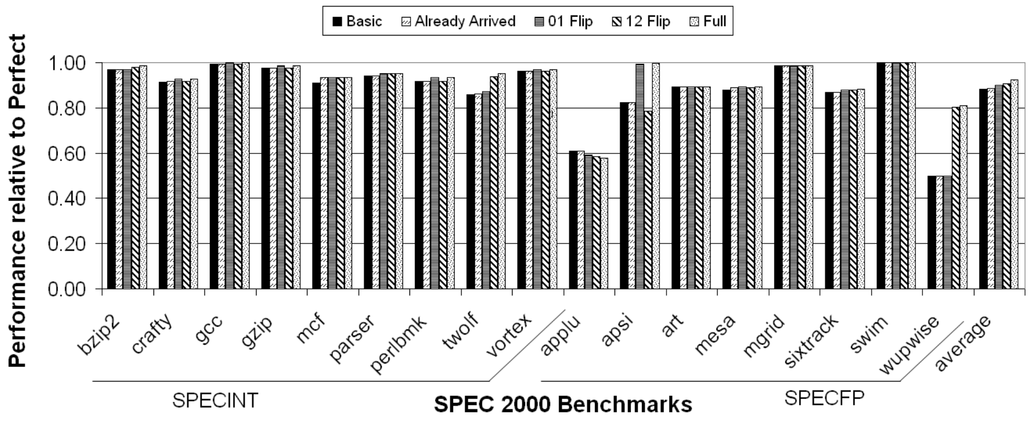


Figure 9. *Effect of Modifications on CDP Performance*

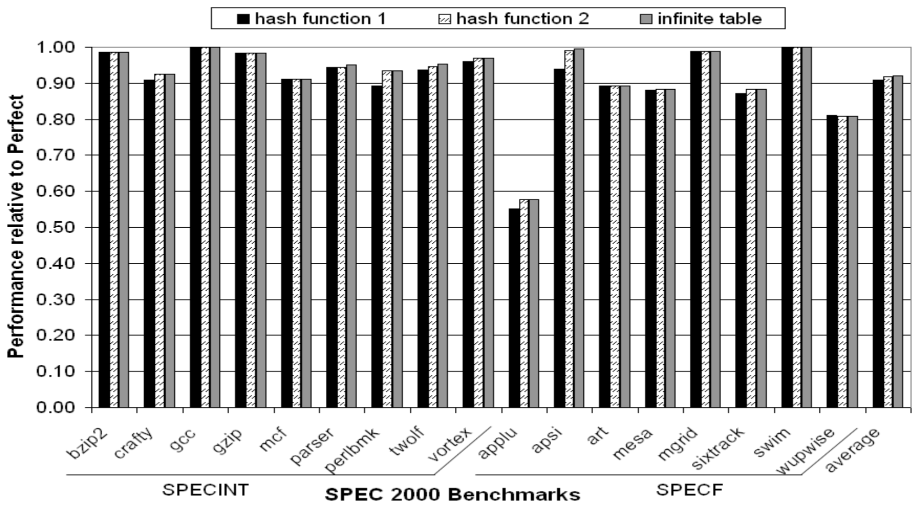


Figure 10. *Reducing Aliasing in CDP Prediction Table*

hold back some loads longer than necessary. However, the 0-1 and 1-2 optimizations interact in a favorable way. Table 2 indicates that on average 4.9% of loads alternate between matching with no, one, or more than one stores. When the 0-1 and 1-2 optimizations are combined, loads that fall into this category are more likely to be deferred long enough to execute safely, without having to wait for all older stores to complete.

The only benchmark that does not benefit from all optimizations is applu. A prediction accuracy breakdown for applu shows that the percentage of loads predicted dependent incorrectly actually increases, even for the 0-1 optimization. This result may be because when the 0-1 optimization prevents a load predicted one-store from being deferred because no block with the PC of the previously offending store's block is in flight, it does not update the predictor table to aggressive, causing a later instance of that load to be more likely to be mispredicted one-store or even conservative.

Because predictions depend on the current state of the entry in the predictor table to which loads index, and because these states are updated based on the accuracy of predictions, the CDP is sensitive to aliasing. Since some of the resulting mispredictions can be costly, aliasing is fundamentally

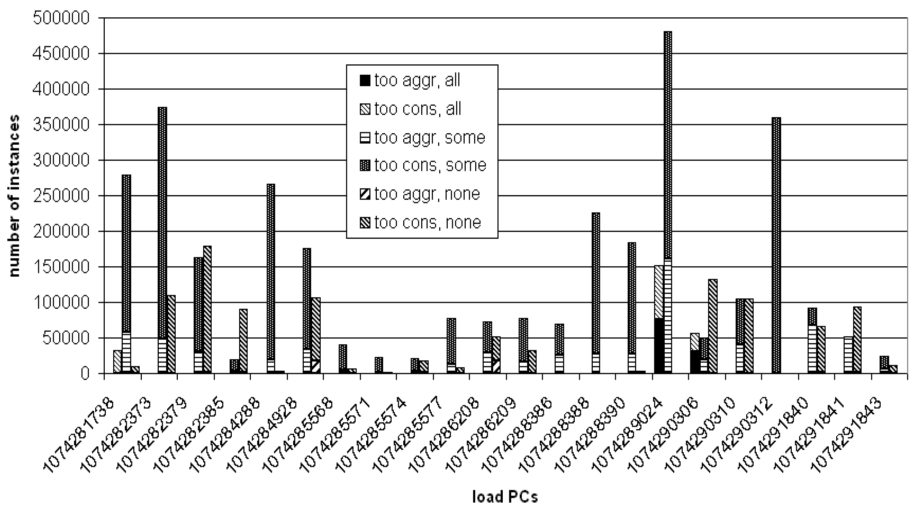


Figure 11. *Comparison of Store Sets and CDP Performance on applu*

more expensive for the CDP that for other predictors. I thus tested CDP performance using different hash functions to index into the predictor table. The results of these experiments are shown in Figure 10. The baseline, indexing into the table with $(LoadPC \bmod TableSize)$, where table size in these experiments is 1024, achieves 90.9% of ideal performance. Using $(LoadPC \bmod (TableSize - 1))$ (hash function 2 in Figure 10) improves performance to 91.9% of ideal, nearly the performance of an infinitely large table (92.0%). Unless $TableSize$ or $(TableSize - 1)$ is a prime number [6], it may not be possible to efficiently compute this modulus in hardware. For all other experiments discussed above, I thus used a table size of 1031 and the hash function $(LoadPC \bmod TableSize)$, which achieves performance essentially identical to that of an infinitely large table.

6.6 CDP and Store Sets Comparison Studies

The benchmark for which the CDP performs worst compared to Store Sets is applu. The graph in Figure 11 shows a comparison of Store Sets and CDP performance on applu. For each load PC, there are three bars, representing how many stores from the in-flight store set actually matched with the load (all, some, or none). Each of these bars in split between instances that the CDP mispredicted too conservatively or too aggressively. The most dominant category is loads for which Sets Sets had some of the stores from the in-flight store set matching and that the CDP predicted overly conservatively. Similar graphs for other benchmarks for which Store Sets performs better show the same trend. This confirms the hypothesis that Store Sets outperforms CDP primarily in the cases where only a few (more than one) of the in-flight stores match: Store Sets waits only for all of the stores in the store set to complete and then issues the load, while the CDP must avoid a violation by forcing the load to wait for all older stores to complete.

It may thus be beneficial to add states to the CDP that allow predictions of some number of stores between one and all. The graph in Figure 12, however, shows that this modification would not be straightforward. The graph shows a breakdown of Store Sets behavior (how many stores were in the in-flight store set, and how many of those stores actually matched the load's target address) for instances in which the CDP made an overly conservative prediction. Most of the data is clustered

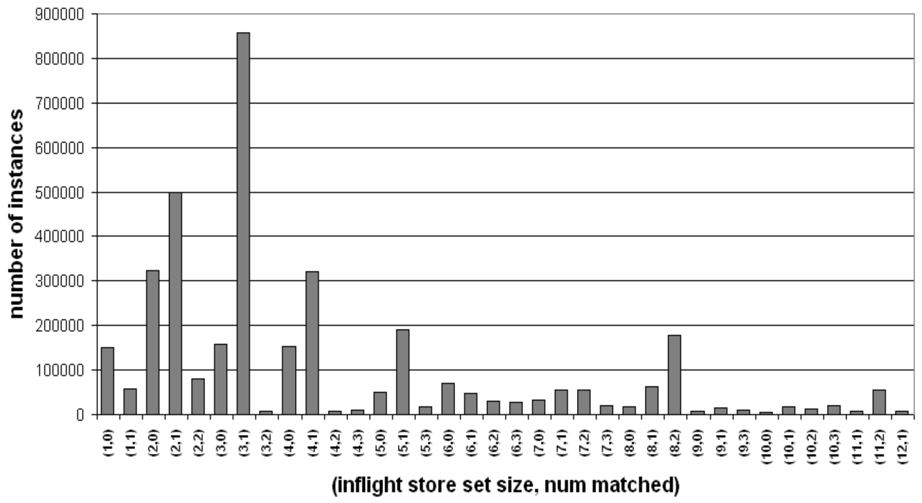


Figure 12. *Breakdown for Store Sets Behavior for CDP's Overly Conservative Loads for applu*

around the lower part of the graph (small in-flight store sets) but ranges up to large in-flight store sets with a variable number of actual store matches. This variability, which is also seen in other benchmarks for which Store Sets outperform the CDP, makes it difficult for the CDP to predict anything more specific than the common cases of zero, one, or many matches.

7 Conclusions

Previously proposed dependence predictors, such as those of Moshovos and Sohi [12], Store Sets [2], and the ECP [25] worked well for centralized superscalar processors, and were shown to be near ideal. Future architectures, however, will be heavily distributed, making it difficult to observe the single, ordered fetch stream and centralized execution information required for these and similar designs.

This thesis evaluates a new type of dependence predictor, which waits for some number of matching stores or other local events to complete before allowing a load to issue. The main advantage of this scheme is that the prediction mechanism is decoupled from reliance on observation of the fetch and execution streams. The best configuration achieves 92% of oracular performance, in an instruction window of up to 2,048 instructions with up to 512 loads or stores.

The simplicity of the CDP allows it to be easily implemented in a distributed microarchitecture. Despite its simplicity, it significantly outperforms another policy, Load-Wait, which is as easy to distribute. Predictors similar to Store Sets and the Exclusive Collision Predictor require access to a centralized fetch stream and global execution information. CDPs use only information that is easily made available locally, yet still achieve good performance.

CDPs may be at a disadvantage when loads are not consistently dependent on the same number of stores. They are also sensitive to overly conservative prediction, either by predicting an independent load to wait on one store match that never arrives, or by predicting a load to be conservative, which prevents violations, but will make any load wait longer than it needs to (no load is dependent on all older stores). CDPs perform well when the number of store matches is small and consistent, since loads are made to wait only as long as necessary. Wakeup conditions can easily be changed under the CDP framework by altering the definition of the wakeup triggering event, such as by including already arrived stores in the count of matches.

As window size increases, resulting in more memory instructions in flight, loads alias with more in-flight stores, and the disadvantages of each dependence prediction scheme are aggravated. Mispredictions become more costly, so accurate prediction becomes more important. Efficient distributed implementations also become more important as larger window sizes increase the burden on the distributed protocol. By slightly complicating the CDP design, such as by allowing predictions of some number of store matches between one and all, or by using path-based or other information to address fluctuating numbers of store matches, CDP performance may be improved further for large windows without compromising its ability to support fully distributed execution.

8 Acknowledgements

I thank Doug Burger for his tremendous support and patience throughout my research experience these past two years. I am also extremely grateful to Simha Sethumadhavan for his guidance, and to him and the rest of the CART group for answering my endless questions. Finally, I extend my thanks to Steve Keckler for his feedback and to Keshav Pingali for his involvement as external committee member.

Robert McDonald first proposed a variant of the CDP. This research is supported by the Defense Advanced Research Projects Agency under contract F33615-01-C-4106, by NSF CISE Research Infrastructure grant EIA-0303609, and by the Intel Undergraduate Research Fellowship for 2007-2008.

References

[1] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the End of Silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[2] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 142–153, 1998.

[3] Compaq Computer Corporation. *Alpha 21264 Microprocessor Hardware Reference Manual*, July 1999.

[4] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[5] E. Ipek, M. Kirman, N. Kirman, and J. F. Martínez. Core Fusion: Accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th International Symposium on Computer Architecture*, pages 186–197, 2007.

[6] M. Kharbutli, K. Irwin, Y. Solihin, and J. Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *Proceedings of the 10th International Conference on High-Performance Computer Architecture*, pages 288–299, 2004.

[7] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, 2002.

[8] C. Kim, S. Sethumadhavan, M. S. S. Govidan, N. Ranganathan, D. Gulati, S. W. Keckler, and D. Burger. Composable lightweight processors. In *Proceedings of the 40th International Symposium of Microarchitecture*, 2007.

[9] A. Moshovos. Personal Communication, November 2007.

[10] A. Moshovos and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th International Symposium on Computer Architecture*, pages 181–193, 1997.

[11] A. Moshovos and G. S. Sohi. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of the 30th International Symposium on Microarchitecture*, pages 235–245, 1997.

[12] A. Moshovos and G. S. Sohi. Speculative memory cloaking and bypassing. *International Journal of Parallel Programming*, pages 427–456, 1999.

[13] S. Onder. Effective memory dependence prediction using speculation levels and color sets. In *Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 232–, 2002.

[14] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.

[15] S. Sethumadhavan, F. Roesner, J. S. Emer, D. Burger, and S. W. Keckler. Late-binding: Enabling unordered load-store queues. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, 2007.

[16] T. Sha, M. M. Martin, and A. Roth. Scalable store-load forwarding via store queue index prediction. In *Proceedings of the 38th International Symposium on Microarchitecture*, pages 159–170, 2005.

[17] T. Sha, M. M. Martin, and A. Roth. NoSQ: Store-Load Communication without a Store Queue. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 106–113, 2006.

[18] T. Sherwood, E. Perelman, and B. Calder. Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 3–14, 2001.

[19] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 414–425, 1995.

[20] J. G. Steffan, C. Colohan, A. Zhai, and T. C. Mowry. The STAMPede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, 2005.

[21] S. S. Stone, K. M. Woley, and M. I. Frank. Address-indexed memory disambiguation and store-to-load forwarding. In *Proceedings of the 38th International Symposium on Microarchitecture*, pages 171–182, 2005.

[22] S. Subramaniam and G. H. Loh. Fire-and-Forget: Load/Store Scheduling with No Store Queue at all. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 273–284, 2006.

[23] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 291–302, December 2003.

[24] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: RAW machines. *IEEE Computer*, 30(9):86–93, September 1997.

[25] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proceedings of the 26th International Symposium on Computer Architecture*, pages 42–53, May 1999.